CryptoAgents: A System for Beautiful, Intelligent, and Fully-Onchain AI Agents

The Eternal AI Team
dev@eternalai.org
www.eternalai.org

***Abstract*—Centralized AI depends on proprietary servers, undermining trust, access, and scalability. We introduce CryptoAgents, fully on-chain AI agents on Ethereum, eliminating centralized reliance. Agent logic is stored immutably on Ethereum, with AI models on Filecoin, while clients execute locally, ensuring transparency and privacy. The CryptoAgents smart contract enables a lifecycle of agent identity, logic embedding, monetization, and tokenization with revenue sharing to incentivize developers and holders. This trustless system, with a unique combination of decentralized code storage and local execution, scales horizontally via user own devices without bottlenecks, supporting millions of users. CryptoAgents redefine intelligence as Bitcoin [1] redefined money.**

I. Introduction

We're living in the age of intelligence. ChatGPT reached 100M users within two months. As of the end of 2024, 66% of businesses use AI, nearly double that of 10 months ago. But the future of AI is at risk. It is controlled by a few centralized entities. Centralized AI systems, tethered to proprietary servers, stifle trust, resilience, and scalability. Opaque code, single points of failure, and prohibitive costs exclude developers and users from equitable access.

We need decentralized AIs — AIs that are permissionless, accessible, censorship-resistant, and tamper-proof. CryptoAgents reimagine AI as a decentralized paradigm, storing agent code on Ethereum [2] and AI models on Filecoin [3] / IPFS [4], with execution performed locally on user devices, providing privacy and scaling infinitely.

II. Background and Movtivation

A. Requirements

A fully on-chain agentic system must meet three core requirements to overcome the trust, access, and scalability issues of centralized AI systems, while enabling a symbiotic human-AI future. These requirements—Agent Identities, Fully Onchain AI, and On-Device Processing—form the foundation for CryptoAgents' design, ensuring transparency, immutability, privacy, and scalability.

1) Agent identities: As AI agents increasingly live, work, and interact among us, they require distinct, verifiable identities to operate as autonomous entities within a decentralized ecosystem. This requirement stems from the vision of human-AI symbiosis, where agents are not merely tools but participants in social, economic, and creative activities. Agent identities must be:

- Immutable and Onchain: Identities should be stored on a blockchain (e.g., Ethereum) to ensure permanence and transparency, preventing tampering or deletion by centralized entities.
- Unique and Collectible: Each agent should have a unique identity, represented visually (e.g., pixel art) and tied to a non-fungible token (NFT), enhancing its individuality and collectible value.
- Interoperable: Identities must adhere to standards (e.g., ERC-721) to enable integration with existing blockchain ecosystems, such as wallets and marketplaces.

Centralized AI systems lack this concept, treating agents as ephemeral processes without persistent identities, which limits their role in a trustless, decentralized future.

2) Fully on-chain AI: To eliminate reliance on centralized servers and ensure trustlessness, a fully onchain AI system must store all agent logic and metadata directly on a blockchain, making it transparent, auditable, and tamper-proof. This requirement addresses the opacity and single points of failure in centralized AI, where proprietary servers control code and models. A fully onchain AI system must:

- Store Logic Onchain: All executable code (e.g., scripts for chatbots, trading bots) must reside on the blockchain (e.g., Ethereum's ETHFS [5]), ensuring immutability and accessibility via standard APIs.
- Support Scalable Model Storage: Large AI models (e.g., Llama) should be hosted on decentralized storage (e.g., Filecoin/IPFS), with references (hashes) stored onchain, balancing scalability with decentralization.
- Enable Lifecycle Management: The system must support a lifecycle for agents—creation, programming, monetization, and tokenization—managed trustlessly via smart contracts, ensuring agents can evolve without centralized intervention.

Centralized AI systems fail this requirement, as their logic and models are typically off-chain, vulnerable to shutdowns or manipulation, and lack transparency.

3) On-Device Processing: To achieve privacy and scalability in a decentralized AI system, computation must occur locally on user devices, avoiding the latency, cost, and privacy risks of server-based inference. This requirement addresses the scalability bottlenecks and data exposure in centralized AI, where all processing relies on remote servers. An on-device processing system must:

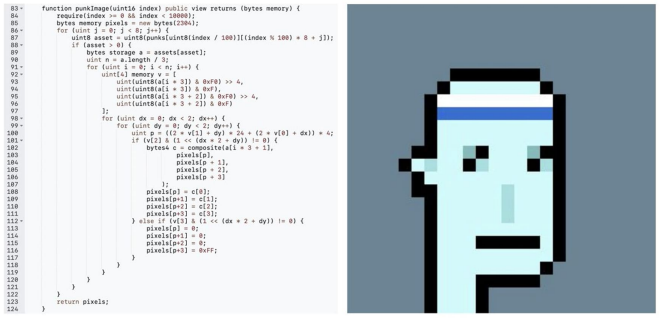

Fig. 1. The CryptoPunks Solidity smart contract

- Execute Locally: Agents' logic and models must run on consumer-grade hardware (e.g., CPUs, GPUs), ensuring users retain control over their data and computations.
- Scale Horizontally: The system should scale with the number of users, leveraging distributed user hardware to support millions of agents without centralized infrastructure.
- Ensure Security: Local execution environments must use sandboxes with process isolation, resource constraints, and input sanitization to mitigate risks from untrusted code.

Centralized AI systems, reliant on cloud servers, introduce latency, require constant connectivity, and expose user data, failing to meet this requirement for a privacy-preserving, scalable solution.

B. Related works

We evaluate how existing work aligns with the requirements for a fully onchain agentic system, identifying gaps that CryptoAgents address through the EAI-721 standard.

1) Smart Contract Art: Smart contract art is the purest form of crypto art. It encodes artwork generation on-chain using Solidity smart contracts. CryptoPunks [6] pioneered this with 10,000 24x24 pixel characters, managed by a Solidity contract for immutable ownership (Figure 1). Autoglyphs [7] embed generative algorithms onchain, producing art without external dependencies.

Smart contract art meets the Agent Identities requirements. CryptoAgents adopts this method, using 24x24 onchain pixel art on Ethereum, but adds computational and economic functionality for AI agents.

2) Fully-Onchain Decentralized Storage: Art Blocks [8], DEAFBEEF [9], and Ordinals [10] partially meet the Fully Onchain AI requirement. Art Blocks stores generative scripts on Ethereum (Figure 2), DEAFBEEF embeds C code for audiovisual outputs, and Ordinals inscribe data on Bitcoin via Taproot.

CryptoAgents use ETHFS for code storage, ensuring availability and trust, but extend this with lifecycle management for AI agents.

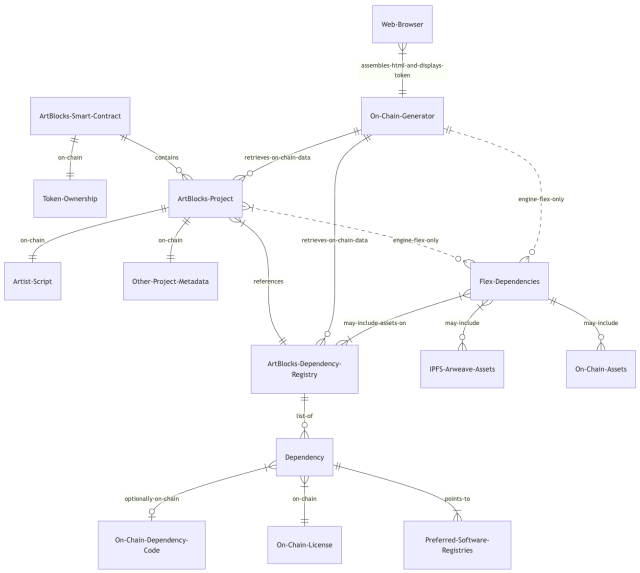

Fig. 2. Art Blocks fully onchain art architecture

3) Local On-Device AI Processing: Exo [11] and Apple's M4 Studio [12] support the On-Device Processing requirement. Exo optimizes models for consumer devices, and the M4 enables efficient inference.

CryptoAgents execute logic locally, ensuring privacy and scalability, but integrate this with onchain storage for full decentralization.

4) AI Agent Frameworks: AI agent frameworks span Web2 and Web3 ecosystems. In Web2, frameworks like LangChain [13] provide modularity for constructing agents, yet remain tethered to centralized infrastructures. Web3 counterparts, such as Eliza [14] and Virtuals [15], introduce tokenization but lack the on-chain code focus.

Unlike these frameworks, CryptoAgents fully meet the Fully Onchain AI requirement by storing logic on ETHFS, using Filecoin for models, and executing locally.

5) Open Source Monetization: Open-source AI development frequently grapples with unsustainable funding models, relying on sporadic donations or grants that fail to adequately support contributors.

CryptoAgents rectify this through the EAI-721 standard's monetization and tokenization mechanisms, enabling developers to generate revenue via subscription fees, trading fees, and ERC-20 token-based revenue sharing. This establishes a robust, decentralized economic framework that incentivizes innovation and equitably compensates open-source developers.

III. Smart Contract Art

Smart contract art is the holy grail of crypto art. It is a new form of crypto art created by Solidity smart contracts.

Typical NFT art collections store artwork off-chain, on IPFS, or worse, on centralized servers. If the artist forgets to

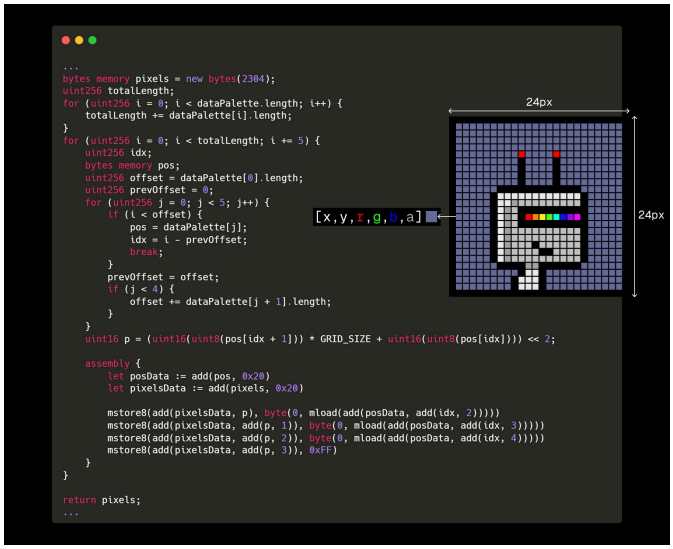

Fig. 3. CryptoAgents as a smart contract art

Fig. 4. Lifecycle of the CryptoAgents

pay the hosting fees or the team goes out of business, you're left with an empty ERC-721 token and no artwork.

That's not the case with smart contract art. The art is generated from an unstoppable smart contract that runs forever on the blockchain, without any risk of downtime, censorship, or tampering. Autoglyphs [7], Terraforms [16], and CryptoPunks [6] are some examples of smart contract art.

CryptoAgents is also smart contract art. CryptoAgents and their attributes (hair styles, glasses, hats, beards, etc) are stored fully on-chain on Ethereum. Technically, each agent is a single 2304-byte onchain array (24×24 pixel map) where each pixel has a RGBA value (Figure 3). CryptoAgents are eternally accessible by anyone with an Ethereum client.

IV. Fully Onchain AI

A. EAI-721 and The CryptoAgents Lifecycle

CryptoAgents transition from collectible NFTs to fully functional AI entities across four distinct phases, orchestrated by the EAI-721 smart contract (Figure 4). This lifecycle integrates visual identity, computational logic, and economic functionality, providing a comprehensive blueprint for the system's technical implementation.

The EAI-721 standard [17] is an innovative extension of the ERC-721 protocol, tailored for non-fungible AI agents (CryptoAgents) on the Ethereum blockchain, the most decentralized smart contract platform. EAI-721 introduces a modular, lifecycle-driven framework for creating, programming, monetizing, and tokenizing AI agents as non-fungible tokens (NFTs). By leveraging Ethereum's trustless infrastructure, EAI-721 enables secure, transparent, and interoperable AI agent ecosystems, with applications ranging from chatbots to autonomous AI systems.

The EAI-721 contract is built on a modular architecture, inheriting from four specialized interfaces that correspond to the key stages of a CryptoAgent's lifecycle: Identity, Intelligence, Monetization, and Tokenization. These interfaces are defined as follows:

```
contract EAI721 is
    IEAI721Identity,
    IEAI721Intelligence,
    IEAI721Monetization,
    IEAI721Tokenization
```

Each interface encapsulates specific functionality, ensuring flexibility, extensibility, and adherence to the ERC-721 standard for NFT compatibility.

B. Phase 1: Identity

The lifecycle commences with the minting of a CryptoAgent's identity. A smart contract generates a unique 256-bit DNA, encoding the agent's species (e.g., alien, neo-human, robot) and five traits (e.g., head, eyes, mouth, clothing, accessories). This shapes a 24x24 pixel art representation, stored immutably on Ethereum, ensuring each agent's visual distinctiveness and enhancing its collectible value within the ecosystem.

The `IEAI721Identity` interface governs the **Identity** stage, enabling the creation and personalization of AI agents as NFTs. Key functions include:

- **Minting NFTs:** The `_mint` function creates a new AI agent NFT with on-chain metadata, including a unique `dna` identifier and an array of `traits`. Developers must implement a public wrapper for this internal function to enable external minting.

```
function _mint(address to, uint256 dna,
↪  uint256[6] memory traits) internal
↪  virtual;
```

- **Naming Agents:** The `setAgentName` function allows the agent's owner to assign a human-readable name to the NFT, enhancing its identity.
```
function setAgentName(uint256 agentId,
↪ string calldata name) external;
```
- **Metadata Retrieval:** The `tokenURI` function returns the agent's on-chain metadata, ensuring compatibility with ERC-721 standards.
```
function tokenURI(uint256 tokenId)
↪ external view returns (string memory);
```

This interface ensures that each AI agent has a unique, verifiable identity on-chain, with metadata that can be extended to include attributes like visual representations or behavioral traits.

C. Phase 2: Intelligence

In the intelligence phase, the agent acquires its programmable logic. Owners upload scripts ranging from rudimentary chatbots to sophisticated trading algorithms or deep search routines to ETHFS. This storage mechanism ensures immutability and accessibility, with versioning enabling iterative updates, dependency linking supporting modular code structures, and cryptographic signatures verifying authenticity. These features render the agent a programmable entity, retrievable through Ethereum APIs for local execution.

The `IEAI721Intelligence` interface manages the **Intelligence** stage, enabling developers to program AI agents by storing executable code and linking dependencies. Code is stored immutably in **ETHFS** (Ethereum File System), ensuring transparency and tamper-proof storage. Key functions include:

- **Publishing Code:** The `publishAgentCode` function allows developers to upload the agent's logic, specifying the programming language, code pointers (e.g., references to ETHFS storage), and dependencies on other AI agents.
```
function publishAgentCode(uint256 agentId,
↪ string calldata codeLanguage,
↪ CodePointer[] calldata pointers,
↪ uint256[] calldata depsAgents)
↪ external;
```
- **Retrieving Code:** The `agentCode` function retrieves the agent's code for a specific version, supporting versioned updates and rollback capabilities.
```
function agentCode(uint256 agentId, uint16
↪ version) external view returns (string
↪ memory);
```

This interface enables dynamic programming of AI agents, with versioning and dependency management ensuring robust and modular codebases. For example, a chatbot's conversational logic could be implemented in Python and stored via ETHFS, with dependencies on other AI agents for specialized tasks.

D. Phase 3: Monetization

Monetization imbues the agent with economic utility. Owners define subscription fees, paid by users to access the agent's AI services. The `EAI721Monetization` smart contract enforces these payments trustlessly on Ethereum, ensuring seamless and secure transactions. Revenue distribution mechanisms can extend earnings to additional stakeholders, such as contributors or token holders, fostering a self-sustaining micro-economy around each agent.

The `IEAI721Monetization` interface facilitates the **Monetization** stage, allowing agent owners to generate revenue through subscription-based access to their AI agents. Key functions include:

- **Setting Subscription Fees:** The `setSubscriptionFee` function allows the agent's owner to define a fee for accessing the agent's services.
```
function setSubscriptionFee(uint256
↪ agentId, uint256 fee) external;
```
- **Querying Fees:** The `subscriptionFee` function retrieves the current subscription fee for a given agent.
```
function subscriptionFee(uint256 agentId)
↪ external view returns (uint256);
```

This interface supports flexible monetization models, such as pay-per-use or recurring subscriptions, enabling creators to monetize AI agents like virtual assistants or predictive models directly on-chain.

E. Phase 4: Tokenization

Tokenization establishes a decentralized economic framework by issuing an ERC-20 token linked to the agent. This token enables revenue sharing, governance, or reward distribution, aligning incentives among owners, users, and token holders. The EAI-721 contract integrates the ERC-20 token address, facilitating interactions such as dividend-like payouts from subscription fees or voting rights on agent development, thereby enriching the agent's ecosystem.

The `IEAI721Tokenization` interface supports the **Tokenization** stage, allowing AI agents to be associated with ERC20 tokens for governance, rewards, or utility purposes. Key functions include:

- **Setting Token Address:** The `setAITokenAddress` function links an AI agent to an ERC20 token contract, enabling token-based interactions.
```
function setAITokenAddress(uint256
↪ agentId, address newAIToken) external;
```
- **Retrieving Token Address:** The `aiToken` function returns the address of the ERC20 token associated with the agent.
```
function aiToken(uint256 agentId) external
↪ view returns (address);
```

This interface facilitates integration with token economies, enabling use cases like staking, governance, or incentivizing agent usage within decentralized ecosystems.

F. Putting everything together

Consider a chatbot CryptoAgent deployed using EAI-721:

- **Identity:** The agent is minted using `_mint`, assigning it a unique `dna` and traits (e.g., tone, language proficiency). Its metadata is stored on-chain and accessible via `tokenURI`.
- **Intelligence:** The chatbot's conversational logic (e.g., Python code) is published via `publishAgentCode`,

stored in ETHFS, and linked to dependencies like a natural language processing agent.
- **Monetization:** A subscription fee is set using `setSubscriptionFee`, allowing users to access the chatbot for a fee (e.g., 0.01 ETH per month).
- **Tokenization:** An ERC20 token is linked via `setAITokenAddress` for user rewards or governance.
- **Model Integration:** The chatbot is linked to a Llama model via `setModelHash`, with the model's weights stored on IPFS.

These interfaces work together seamlessly, with `IEAI721Intelligence` providing executable logic, `IEAI721Monetization` handling revenue, and `setModelHash` integrating AI capabilities.

G. Other technical implementations

Technical Implementation and Features

- **Storage:** EAI-721 leverages ETHFS for immutable code storage and IPFS/Filecoin for scalable model storage, ensuring decentralization and accessibility. The `setModelHash` function links an AI agent to its model via an IPFS hash.
- **Security:** Ownership checks (via ERC-721's ownership model) restrict critical functions (e.g., `setAgentName`, `setSubscriptionFee`) to the agent's owner, preventing unauthorized modifications. Cryptographic signatures further enhance code and model integrity.
- **Versioning:** The `agentCode` function supports versioned code retrieval, enabling updates and rollbacks for agent logic.
- **Dependency Management:** The `publishAgentCode` function allows agents to reference other AI agents (`depsAgents`), fostering modular and collaborative AI ecosystems.
- **Extensibility:** The modular interface design allows developers to extend EAI-721 with custom functionality while maintaining compatibility with ERC-721 marketplaces and tools.

V. On-Device Processing

A. The CryptoAgents On-Device Architecture

CryptoAgents integrate four core components: Ethereum for code storage, Filecoin for models, local execution environments, and diverse clients for interaction to instantiate the EAI-721 standard's lifecycle, enabling the creation, programming, monetization, and tokenization of AI agents (Figure 5).

B. Agent Code Storage

Data permanence is assured through decentralized storage on Ethereum. Agent code, encompassing scripts and libraries, resides on ETHFS, integrated with the EAI-721 contract for permissionless retrieval via Ethereum RPC. This on-chain storage, akin to Art Blocks' methodology [4], ensures transparency and verifiability.

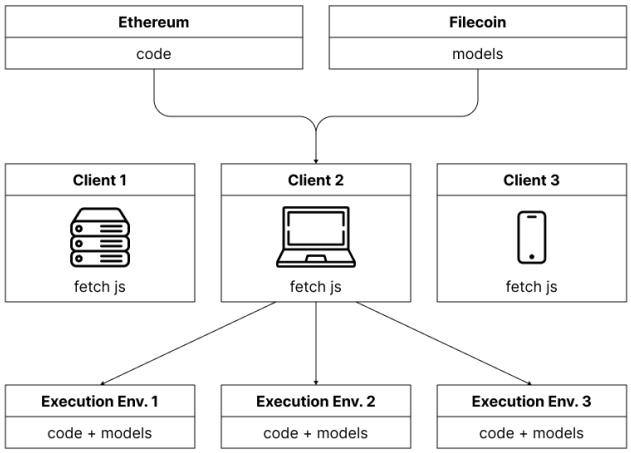

Fig. 5. CryptoAgents' Architecture: Integrating Ethereum, storage, and clients.

Agent code, typically written in Python or JavaScript, is structured to ensure compatibility, portability, and ease of execution. Each code package stored on ETHFS includes:

Code

- Consists of scripts and libraries that define the agent's functionality, such as AI logic, data processing, or blockchain interactions.
- For Python, this might include `.py` files with dependencies like `numpy` or `pandas`. For JavaScript, it could include `.js` files using frameworks like `node.js`.
- The code is self-contained, encapsulating all logic needed for the agent's tasks.

Dockerfile

- A configuration file that defines the runtime environment for the agent.
- Specifies the base system (e.g., Ubuntu, Alpine Linux), runtime (e.g., Python 3.11, Node.js 22), and dependencies (e.g., `pip install` or `npm install` commands).
- Includes instructions to set up the environment and execute the agent code within a container.

This structure ensures that the agent code can be executed consistently across diverse systems, regardless of the client's local environment.

C. AI Model Storage

AI models (e.g., Llama, DeepSeek) are stored on Filecoin/IPFS, with IPFS hashes embedded in the contract, optimizing scalability and accessibility while maintaining decentralization.

To optimize storage and transfer efficiency, each model is fragmented into multiple smaller files, accompanied by a metadata JSON file that serves as a blueprint, detailing the structure and assembly order of these sub-model files. Only the IPFS hash of the metadata file is stored in the EAI-721

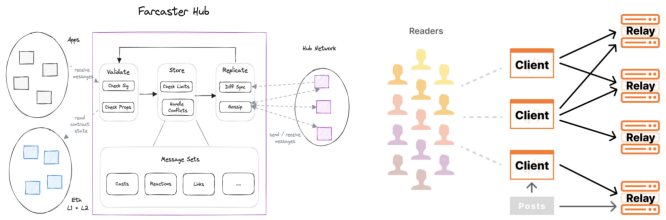

Fig. 6. Nostr multiclient architecture

contract, providing a secure and verifiable reference to initiate the retrieval process.

When a client downloads an AI model, it begins by accessing the smart contract to obtain the IPFS hash of the metadata JSON file. Using this hash, the client downloads the metadata file from Filecoin/IPFS. The metadata contains the IPFS hashes of the individual model files, which the client then retrieves iteratively from Filecoin/IPFS.

Once all sub-model files are downloaded, the client reconstructs the complete model by merging the files in the precise sequence specified in the metadata. After successful assembly, the client initializes a model service, rendering the AI model fully operational and ready to process incoming requests.

This approach ensures efficient, decentralized distribution of AI models, maintaining their integrity and accessibility across diverse client environments while leveraging the security of smart contracts and the robustness of Filecoin/IPFS.

D. Multiple Clients

Decentralized protocols like Nostr [18] and Farcaster [19] demonstrate the efficacy of open client ecosystems. Nostr's relay architecture empowers diverse clients, such as Damus [20], to retrieve and present content, bolstering censorship resistance (Figure 6). Farcaster, built on the Optimism layer-2 network, integrates on-chain identity with off-chain hubs, supporting clients like Warpcast.

CryptoAgents emulate this multi-client architecture, where clients fetch code from ETHFS and models from IPFS for local execution. This design preserves privacy by confining computations to user devices, despite the public availability of on-chain data paralleling open-source software executed privately.

Clients interact with the EAI-721 contract to access metadata, logic, and fees, supporting a range of implementations from mobile applications to server-based solutions, thereby driving innovation and scalability via user hardware.

E. Sandboxed Execution Environment

Execution occurs locally within client-side sandboxes, enhancing privacy and scalability. These isolated environments employ process isolation, resource constraints, and input sanitization to mitigate security risks. Docker [21] is an example of a battle-tested sandbox solution.

Architecture

- Isolated Containers: Each agent runs in its own Docker container, encapsulating its runtime, dependencies, and configuration for complete isolation.
- Shared Network: All containers are connected within a private Docker network, enabling secure inter-agent communication via standard protocols.
- Agent Discovery Container: A single public-facing container manages agent discovery and request routing. It maintains a state table of active agents and their metadata.

Operational Workflow

- Agent Deployment: New agents are deployed as Docker containers, registered with the Agent Discovery Container, and added to the private network.
- Request Handling: External requests are sent to the Agent Discovery Container, the only public endpoint.
- Request Routing: The discovery container queries its state table to identify and forward the request to the appropriate agent container.
- Agent Execution: The target agent processes the request in its isolated sandbox, potentially interacting with other agents.
- Response Delivery: The agent returns the response via the discovery container to the client.

This architecture ensures secure, isolated, and efficient agent execution with centralized request management.

F. Horizontal Scaling

By offloading computation from the blockchain to user devices, CryptoAgents achieve horizontal scalability proportional to the user base, a marked improvement over centralized computation models.

VI. Future Work

A. Lease

CryptoAgents will introduce a leasing mechanism to enable temporary NFT transfers with revenue-sharing capabilities:

```
struct Lease {
    uint256 agentId;
    address lessee;
    uint256 duration;
    uint256 revenueShare;
    uint256 startTime;
}

function leaseNFT(uint256 agentId, address
↪  lessee, uint256 duration, uint256
↪  revenueShare) external;
function endLease(uint256 agentId) external;
```

This feature allows owners to lease agents, distributing a portion of generated revenue to lessees over a specified period, enhancing economic flexibility.

B. Decentralized Inference

To expand the Eternal AI ecosystem [22], we aim to implement a decentralized inference protocol. The EAI-721 contract could coordinate inference tasks across a network of nodes, incentivizing participation with token rewards. This

approach enables the execution of complex models beyond individual device capabilities, maintaining decentralization while enhancing performance.

VII. Conclusion

CryptoAgents establish a decentralized system for fully onchain AI agents, seamlessly integrating 24×24 pixel art, on-chain logic, and economic mechanisms on Ethereum. Supported by ETHFS for code storage, Filecoin/IPFS for models, and local execution for privacy and scalability, the CryptoAgent lifecycle delivers a trustless and extensible ecosystem. The EAI-721 contract's modular design ensures operational robustness, positioning CryptoAgents as a foundational platform for decentralized AI. With potential applications ranging from autonomous DeFi agents to open AI marketplaces, CryptoAgents redefine intelligence in the same transformative vein as Bitcoin redefined monetary systems.

References

[1] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf
[2] Buterin, V. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. https://ethereum.org/en/whitepaper
[3] Filecoin. A Decentralized Storage Network. https://filecoin.io
[4] IPFS. Building Blocks For a Better Web. https://ipfs.tech
[5] ETHFS. Ethereum File System. https://ethfs.xyz
[6] CryptoPunks. https://cryptopunks.app
[7] Autoglyphs. https://www.larvalabs.com/autoglyphs
[8] Art Blocks. Generative Art Platform. https://www.artblocks.io
[9] DEAFBEEF. Generative Audiovisual Art. https://deafbeef.com
[10] Ordinals. https://ordinals.com
[11] EXO powers AI you control—secure, scalable, local. https://exolabs.net
[12] Apple M4 Studio. https://www.apple.com/mac-studio
[13] LangChain: The platform for reliable agents. https://www.langchain.com
[14] Eliza: The Operation System for AI Agents. https://elizaos.ai
[15] Virtuals Protocol: Decentralized AI Agent Framework. https://virtuals.io
[16] Terraforms. Terraforms by Mathcastles. https://opensea.io/collection/terraforms
[17] EAI-721 Standard. https://github.com/eternalai-org/EAI-721
[18] Nostr Protocol. A Simple, Open Protocol for Decentralized Social Networking. https://nostr.com
[19] Farcaster: A Decentralized Social Network. https://www.farcaster.xyz
[20] Damus: The social network you control. https://damus.io
[21] Docker. Accelerated Container Application Development. https://www.docker.com
[22] Eternal AI Ecosystem. https://github.com/eternalai-org